Covariance tracking: architecture optimizations for embedded systems

Andrés Romero, Lionel Lacassagne, Michèle Gouiffès, Ali Hassan Zahraee

EURASIP Journal on Advances in Signal Processing

SpringerOpen

Covariance tracking: architecture optimizations for embedded systems

EURASIP Journal on Advances in Signal Processing 2014,
2014:175 doi:10.1186/1687-6180-2014-175

Andrés Romero (andresrommier@gmail.com)
Lionel Lacassagne (lionel.lacassagne@lri.fr)
Michèle Gouiffès (michele.gouiffes@u-psud.fr)
Ali Hassan Zahraee (ahzahraee@gmail.com)

ISSN 1687-6180

Article type Research

Submission date 7 March 2014

Acceptance date 16 October 2014

Publication date 6 December 2014

Article URL http://asp.eurasipjournals.com/content/2014/1/175

Covariance tracking: architecture optimizations for embedded systems

Andrés Romero[1*,†]
*Corresponding author
Email: andres.romero@lri.fr

Lionel Lacassagne[1,†]
Email: lionel.lacassagne@lri.fr

Michèle Gouiffès[2]
Email: michele.guiffes@u-psud.fr

Ali Hassan Zahraee[1]
Email: ali.hassan@u-psud.fr

[1]Laboratoire de Recherche en Informatique, Université Paris-Sud, Bat 650, Université Paris Sud, Orsay, France

[2]Laboratoire d'Informatique pour la Mécanique et les Sciences de l'Ingénieur, Bat 508, Université Paris Sud, Orsay, France

[†]Equal contributors.

Abstract

Covariance matching techniques have recently grown in interest due to their good performances for object retrieval, detection, and tracking. By mixing color and texture information in a compact representation, it can be applied to various kinds of objects (textured or not, rigid or not). Unfortunately, the original version requires heavy computations and is difficult to execute in real time on embedded systems. This article presents a review on different versions of the algorithm and its various applications; our aim is to describe the most crucial challenges and particularities that appeared when implementing and optimizing the covariance matching algorithm on a variety of desktop processors and on low-power processors suitable for embedded systems. An application of texture classification is used to compare different versions of the region descriptor. Then a comprehensive study is made to reach a higher level of performance on multi-core CPU architectures by comparing different ways to structure the information, using single instruction, multiple data (SIMD) instructions and advanced loop transformations. The execution time is reduced significantly on two dual-core CPU architectures for embedded computing: ARM Cortex-A9 and Cortex-A15 and Intel Penryn-M U9300 and Haswell-M 4650U. According to our experiments on covariance tracking, it is possible to reach a speedup greater than $\times 2$ on both ARM and Intel architectures, when compared to the original algorithm, leading to real-time execution.

Keywords

Covariance tracking; SIMD; Multi-core; Embedded systems

Introduction

Tracking consists in estimating the evolution in state (e.g., location, size, orientation) of a moving target over time. This process is often subdivided into two other subproblems: detection and matching. Detection deals with the difficulties of generic object recognition, i.e., finding instances from a particular object class or semantic category (e.g., humans, faces, vehicles) registered in digital images and videos. On the other hand, matching methods provide the location which maximizes the similarity with the objects previously detected in the sequence. Generic object recognition requires models that cope with the diversity of instances' appearances and shapes. This is generally made by learning techniques and classification. Conversely, matching algorithms analyze particular information and construct discriminative models that allow to disambiguate different instances from the same category and avoid confusions.

The main difficulty of tracking is to trace target trajectories and adapt to changes of appearance, pose, orientation, scale, and shape. Since the beginnings of computer vision, a diversity of tracking methods have been proposed, some of them construct path and state evolution estimations using a Bayesian framework (e.g., particle filters, hidden Markov models), others measure the perceived optical flow in order to determine object displacements and scale changes (median flow) [1]. Exhaustive appearance-based methods compare a dense set of overlapping candidate locations to detect the one that fits best with some kind of template or model. When *a priori* information about the target location and its dynamics (e.g., speed and acceleration) is available, the number of comparisons can be reduced enormously by giving preference to the more likely target regions. Other accelerations can be achieved using local searches that are based on gradient-descent algorithms able to handle small target displacements and geometrical changes. Among these approaches, feature points tracking techniques are very popular [2] since points can be extracted in most scenes, contrary to lines or other geometric features. Because they represent very local patterns, their motion models can be assumed as rigid and be estimated in a very efficient way. This method, as well as block matching, are raw-pixel methods, since the target is directly represented by its pixels matrix.

In order to deal with non-rigid motion, kernel-based methods such as mean-shift (MS) [3] and [4] use a representation based on color or texture distribution.

Covariance tracking (*CT*) [5] is a very interesting and elegant alternative which offers a compact target representation based on the spatial correlation of different features computed at each pixel in the target bounding box. Very satisfying tracking performances have been observed for diverse kinds of objects (e.g., with rigid motion or not, with texture or not). CT has been studied extensively, and many feature configurations and arrays of covariance descriptors have been proposed to improve its discrimination power [6-11] and. Smoother trajectories can be obtained by considering target dynamics; therefore, they increase tracking accuracy and reduce the search space [12,13]. Genetic algorithms [14] can also be used to accelerate the convergence towards the optimal solution of the best candidate position, considering a search in a large image. But, to our knowledge, little work has been done to analyze the computational demands of CT and its portability to embedded systems [15]. The goal of this article is to fill this gap, analyze the algorithm's computational behavior for different implementations, and measure their load on embedded architectures. A study is also made to compare different sizes and configurations of the descriptors in terms of discrimination power through a texture classification application.

The article is structured as follows. The first section introduces some of the basic principles of the CT algorithm and provides a brief description of the different searching and matching methods that can be associated with C. Then various configurations of the covariance matrix are evaluated. Finally, we provide an in-depth description of implementation details and suitable acceleration techniques proposed to achieve a higher level of performance. Experiments and details about the algorithm implementation are presented in the final section that comes followed by our conclusions.

1 Covariance matrices as image region descriptors

Let I represent a luminance (grayscale) or a color image with three channels and consider a rectangular region of size $n = W \times H$ (it can be the bounding box of the target to be tracked for example). Let F be the $W \times H \times n_F$ dimensional feature image extracted from I

$$F_{uv} = F(\boldsymbol{p}_{uv}) = \phi(I, \boldsymbol{p}_{uv}) \text{with } \boldsymbol{p}_{uv} = (x_u, y_v) \tag{1}$$

where ϕ is any n_F-dimensional mapping forming a feature vector for each pixel of the bounding box. The features can be spatial coordinates $\boldsymbol{p}_{uv}$, intensity, color (in any color space), gradients, filter responses, or any possible set of images obtained from I. Now, let $\{\mathbf{z}_k\}_{k=1\cdots n}$ be a set of n_F-dimensional feature vectors inside the rectangular region $R \subset F$ of n pixels. Concerning notations, $\boldsymbol{p}_{uv}$ stands for the pixel at uth row and vth column.

The region R is represented with the $n_F \times n_F$ covariance matrix

$$\mathbf{C}_R = \frac{1}{n-1} \sum_{k=1}^{n} (\mathbf{z}_k - \boldsymbol{\mu})(\mathbf{z}_k - \boldsymbol{\mu})^T \tag{2}$$

where $\boldsymbol{\mu}$ is the mean feature vector computed on the n points.

The covariance matrix is a $n_F \times n_F$ matrix which fuses multiple features naturally by measuring their correlations. The diagonal terms represent the variance of each feature, while elements outside this diagonal are the correlations. Thanks to the averaging in the covariance computation, noisy pixels are largely filtered out, which is an interesting advantage when compared to raw-pixel methods. Covariance matrices are more compact than most classical object descriptors. Indeed, due to symmetry, $\mathbf{C}_R$ has only $(n_F^2 + n_F)/2$ different values whatever the size of the target. To some extent, it is robust against scale changes, because all values are normalized by the size of the object, and against rotation when the locations coordinates $\boldsymbol{p}_{uv}$ are replaced by the distance to the center of the bounding box.

The covariance descriptor ceases to be rotationally invariant when orientation information is introduced in the feature vector such as the norm of gradients with respect to x and y directions. The information considered by the covariance descriptor should be adapted to the problem at hand, because they depend on the application, as described in the next paragraph.

1.1 Covariance descriptor feature spaces

Covariance descriptors have been used in computer vision for object detection [16], reidentification [10, 11] and tracking [5]. The recommended set of features to use depends significantly on the application and the nature of the object: tracking faces is different than tracking pedestrians because faces are somehow more rigid than pedestrians which have more articulations. Color is an important hint for pedestrian or vehicle tracking/reidentification because of their clothes or bodywork color. But color is less significant for reidentification or tracking faces because the set of colors they exhibit is relatively limited.

Table 1 displays a summary of the more common feature combinations used by covariance descriptors in computer vision. The most obvious ones are the components from different color spaces such as RGB and HSV. Pixel brightness in the grayscale image I and its local directional gradients as absolute values $|I_x|$ and $|I_y|$, gradient magnitude $\sqrt{I_x^2 + I_y^2}$, and its angle calculated as $\arctan \frac{|I_x|}{|I_y|}$. Foreground images $\mathbf{G}$ resulting from background subtraction methods and its gradients $\mathbf{G}_x$ and $\mathbf{G}_y$. Features $g_{00}(x, y)$ to $g_{74}(x, y)$ represent the 2D Gabor kernel as a product of an elliptical Gaussian and a complex plane wave [9].

Table 1 Features considered by the covariance descriptor depending on the application

Application	Feature set $\phi(\boldsymbol{I}, \boldsymbol{p})$ with $\boldsymbol{p} = (\boldsymbol{x}, \boldsymbol{y})$
Face tracking and recognition [9]	$\left[\begin{array}{ccccccc} x & y & \lvert I_x\rvert & \lvert I_y\rvert & \lvert I_{xx}\rvert & \lvert I_{yy}\rvert \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & I & \lvert I_x\rvert & \lvert I_y\rvert & \lvert I_{xx}\rvert & \lvert I_{yy}\rvert & \theta(x,y) \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & I & g_{00}(x,y) & g_{01}(x,y) & \cdots & g_{74}(x,y) \end{array}\right]$
Pedestrian detection [16,17]	$\left[\begin{array}{ccccccc} x & y & \lvert I_x\rvert & \lvert I_y\rvert & \sqrt{I_x^2+I_y^2} & \lvert I_{xx}\rvert & \lvert I_{yy}\rvert & arctan\frac{\lvert I_x\rvert}{\lvert I_y\rvert} \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & \lvert I_x\rvert & \lvert I_y\rvert & \sqrt{I_x^2+I_y^2} & arctan\frac{\lvert I_x\rvert}{\lvert I_y\rvert} & \mathbf{G} & \sqrt{\mathbf{G}_x^2+\mathbf{G}_y^2} \end{array}\right]$
Pedestrian tracking [5,10,11,16] and [18]	$\left[\begin{array}{ccccccc} x & y & R & G & B & \lvert I_x\rvert & \lvert I_y\rvert \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & R & G & B & \lvert I_x\rvert & \lvert I_y\rvert & \lvert I_{xx}\rvert & \lvert I_{yy}\rvert \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & H & S & V & \lvert I_x\rvert & \lvert I_y\rvert \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & R & G & B & \mathrm{Var}_{\mathrm{LBP}} \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & I & \sin(\mathrm{LBP}_{\theta_0}) & \cos(\mathrm{LBP}_{\theta_0}) & \sin(\mathrm{LBP}_{\theta_1}) & \cos(\mathrm{LBP}_{\theta_1}) \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & R & G & B & \sin(\mathrm{LBP}_{\theta_0}) & \cos(\mathrm{LBP}_{\theta_0}) & \sin(\mathrm{LBP}_{\theta_1}) & \cos(\mathrm{LBP}_{\theta_1}) \end{array}\right]$

Some texture analysis and tracking methods use local binary patterns (*LBP*) in the place of Gabor filters and the reason is that LBP operators are much more simple and economical. Values $\mathrm{Var}_{\mathrm{LBP}}$, LBP_{θ_0}, and LBP_{θ_1} in Table 1 represent local binary pattern variance (which is a classical property of the LBP operator [19]) and the angles defined by them, as detailed in [18]. This version of the feature vector has shown very good performances for tracking, both in terms of robustness and computation times, and requires a far shorter vector ($n_F = 7$) when compared to Gabor filters ($n_F = 43$). In the rest of the paper, for the algorithmic optimization, a vector of five to nine features is considered, but note that the proposed optimizations can be applied to any matrix size.

Now, let us detail the computation of the covariance descriptor.

1.2 Covariance descriptor computation

After some term expansions and rearrangements on Equation 2, the (i, j)-th element of the covariance matrix can be expressed as

$$\mathbf{C}_R(i,j) = \frac{1}{n-1}\left[\sum_{k=1}^{n} z_k(i)z_k(j) - \frac{1}{n}\sum_{k=1}^{n} z_k(i)\sum_{k=1}^{n} z_k(j)\right]. \tag{3}$$

Therefore, the covariance in a given region depends on the sum of each feature dimension $z(i)_{i=1\cdots n}$, as well as the sum of the multiplications of any pair of features $z(i)z(j)_{i,j=1\cdots n}$, requiring in total $n_F + n_F^2/2$ integral images, one for each feature dimension $z(i)$ and one for the multiplication of any pair of feature dimensions $z(i)z(j)$ (the covariance matrix is symmetric).

Let A be a $W \times H \times n_F$ tensor of the integral images of each feature dimension

$$A_{uv}(i) = \sum_{\boldsymbol{p}\in R(11,uv)} F_{uv}(i) \text{for } i = i\cdots n_F, \tag{4}$$

where $R(11, uv)$ is the region bounded by the top-left image corner $\boldsymbol{p}_{11} = (1, 1)$ and any other point in

the image $\boldsymbol{p}_{uv} = (x_u, y_v)$. In a general way, let $R(uv, u'v')$ be the rectangular region defined by the top-left point $\boldsymbol{p}_{uv}$ and the right-bottom point $\boldsymbol{p}_{u'v'}$. Similarly, the tensor containing the feature product-pair integral images is denoted as

$$B_{uv}(i,j) = \sum_{\boldsymbol{p} \in R(11,uv)} F_{uv}(i) F_{uv}(j) \text{for } i,j = i \cdots n_F. \tag{5}$$

Now, for any point $\boldsymbol{p}_{uv}$, let $\mathbf{A}_{uv}$ be a n_F-dimensional vector and $\mathbf{B}$ a $n_F \times n_F$ dimensional matrix such as

$$\mathbf{A}_{uv} = [A_{uv}(1) \cdots A_{uv}(n_F)]^T \text{ and } \mathbf{B}_{uv} = \begin{pmatrix} B_{uv}(1,1) & \cdots & B_{uv}(1,n_F) \\ & \vdots & \\ B_{uv}(n_F,1) & \cdots & B_{uv}(n_F,n_F) \end{pmatrix}. \tag{6}$$

The covariance of the region bounded by $(1,1)$ and $\boldsymbol{p}_{uv}$ is

$$\mathbf{C}_R(11,uv) = \frac{1}{n-1}\left[\mathbf{B}_{uv} - \frac{1}{n}\mathbf{A}_{uv}\mathbf{A}_{uv}^T\right], \tag{7}$$

where n is the number of pixels in the R under investigation. Similarly, and after some algebraic manipulations, the covariance of the region $R(uv, u'v')$ as it was presented in [20] is

$$\begin{aligned} \mathbf{C}_{R(uv,u'v')} = \tfrac{1}{n-1}\Big[&(\mathbf{B}_{u'v'} + \mathbf{B}_{uv} - \mathbf{B}_{u'v} - \mathbf{B}_{uv'}) \\ &-\tfrac{1}{n}\left(\mathbf{A}_{u'v'} + \mathbf{A}_{uv} - \mathbf{A}_{uv'} - \mathbf{A}_{u'v}\right) \cdot \left(\mathbf{A}_{u'v'} + \mathbf{A}_{uv} - \mathbf{A}_{uv'} - \mathbf{A}_{u'v}\right)^T\Big]. \end{aligned} \tag{8}$$

Once the integral images have been calculated, the covariance of any rectangular region can be computed in $O(n_F^2)$ time regardless of the size of the region $R(uv, u'v')$. The complete process is represented graphically in Figure 1, where different image-processing operators are applied to the initial image (top left) to calculate the set of feature images (top right). Each feature component i is used to generate the integral image $A_{uv}(i)$ (bottom left) and the crossed product between features i and j is used to calculate the second order integral images $B_{uv}(i,j)$.

Figure 1 Covariance descriptor computation. The image is first decomposed into an array of feature images (feature image tensor) applying the feature map $F_{uv} = \phi(I, \boldsymbol{p}_{uv})$. Then the crossed-products of these features are computed; using these arrays, the tensor integral images $A_{u'v'}(i)$ and the second order integral images tensor $B_{u'v'}(i,j)$ are computed.

Next section explains the covariance matching process.

2 Searching and matching

Covariance models and instances can be compared and matched using a simple nearest neighbor approach, i.e., by finding the covariance descriptors that best resemble a model. The problem is that covariance matrices and symmetric positive definite (SPD) matrices in general is that they do not lie on the Euclidean space and many common and widely known operations in Euclidean spaces are not applicable or require to be adapted (e.g., a SPD matrix multiplied by a negative scalar is no longer a valid SPD matrix). A $n_F \times n_F$ SPD matrix only has $n_F \times (n_F + 1)/2$ different elements; while it is possible to vectorize them and perform element-by-element subtraction, this approach provides very poor results as it fails to analyze the correlations between variables and the patterns stored in them. A solution to this

problem is proposed in [21] where a dissimilarity measure between two covariance matrices is given as

$$\rho(\mathbf{C}_1, \mathbf{C}_2) = \sqrt{\sum_{i=1}^{n_F} \ln^2 \lambda_i(\mathbf{C}_1, \mathbf{C}_2)} \tag{9}$$

where $\{\lambda_i(\mathbf{C}_1, \mathbf{C}_2)\}_{i=1,\cdots,n_F}$ are the generalized eigenvalues of $\mathbf{C}_1$ and $\mathbf{C}_2$ computed from

$$\lambda_i \mathbf{C}_1 \mathbf{x}_i - \mathbf{C}_2 \mathbf{x}_i = 0 i = 1, \cdots, n_F. \tag{10}$$

The tracking starts in the first frame of the sequence, by computing the covariance matrix $\mathbf{C}_1$ in the bounding box of the target under consideration (i.e., the model). The initial detection is not detailed in this paper since it can be made in various ways, by object recognition or background subtraction for example. The tracking procedure consists in determining the new target positions for the successive frames by comparing the covariance matrix $\mathbf{C}_2$ (i.e., the candidate position) and minimizing the Riemannian distance (9).

Figure 2 depicts two possible searching strategies: the *exhaustive search* approach (left) and a gradient-based local search or *steepest descent* approach (right). Exhaustive search methods uniformly sample a large number of candidate positions scanning the whole image (or the region surrounding the previous target position). Steepest descent methods look for the position which maximizes the appearance similarity when compared to the target model. Gradient-based methods do not require a large number of matrix comparisons; however, they do require to run iteratively until convergence, causing their computation time to be very unpredictable. Another limitation (and probably the most important one) is that contrary to exhaustive search approaches, local search may fail for targets undergoing brutal motion or target occlusions. The reason behind this problem is that local search methods tend to fall into local minima. Due to these limitations, the exhaustive search method was preferred for the tracking application implemented for this research.

Figure 2 Illustration of two possible covariance tracking *search* strategies. The *exhaustive search* approach (left) where a large number of candidate bounding boxes is uniformly sampled and evaluated, and the *steepest descent* method (right) where gradient-based *local search* is launched looking for the position that minimizes the Riemannian distance.

3 Feature vector evaluation

The objective of this section is to determine the most discriminative vector combination and the ideal number of features (n_F) to use. Multiple feature combinations were tested using a texture classification method. The *KTH-Tips* dataset [22] is composed of ten different texture classes each one represented by 81 different samples of size 200×200 taken at different scales, illuminations, and poses.

There are different approaches for texture classification with covariance matrices. Most methods subdivide the image in small overlapping subregions and compute a descriptor associated to each one. The drawback with this approach is that it increases the number of matrix comparisons and the storage required. To avoid this problem, the local log-Euclidean covariance matrix (L2ECM) [6] computes a single covariance matrix from the log-Euclidean transformations of other (simpler) covariance matrices calculated at every pixel neighborhood (typical sizes are 15×15 or 30×30). While L2ECM provides high texture reidentification scores, its main drawback is that it considerably increases the number of computations and the memory space that is required during the computation phase. Therefore, L2ECM is clearly not appropriate for embedded platforms; hence, for the sake of simplicity, we were much more

inclined to use a very simple approach and compute a single covariance descriptor for every sample and feature combination.

Ten random images were selected for the training of each texture class (from the set of 81 samples that represents each one); the remaining samples were used during the classification evaluation. The descriptor obtained from each test image is compared against all the covariance matrices inside the different training sets (ten samples for each texture class) using the Riemannian metric proposed in [20]. A label is assigned to each class using the KNN[a] algorithm counting the number of votes of each texture class for the closest $k = 5$ samples. The same procedure was repeated ten times to summarize and avoid unstable or misguiding results.

To evaluate the quality of our classification results, we counted the number of true/false positives and negatives and calculated their associated F_1 score (this represents the weighted average of the precision and recall) defined as

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}, \tag{11}$$

where

$$\begin{aligned} \text{precision} &= \frac{\#\text{True positives}}{\#\text{True positives}+\#\text{False positives}} \\ \text{recall} &= \frac{\#\text{True positives}}{\#\text{True positives}+\#\text{False negatives}}. \end{aligned} \tag{12}$$

Multiple feature combinations were evaluated based on the spatial coordinates (x and y), the luminance (I) and color channels (R, G, and B), the first and second order gradient magnitudes ($|I_x|$, $|I_y|$, $|I_{xx}|$, $|I_{xy}|$ and $|I_{yy}|$), and the enhanced local binary covariance matrices (ELBCM) features proposed in [18].

Figure 3 depicts the set of feature combinations that were evaluated and their associated F_1 scores. Each combination has a set of points representing the score associated to each one of the different texture classes. Boxplots are used to highlight their concentration and their median (depicted by the horizontal bars in pink inside the boxes). The figure is divided in two rows for grayscale and color-based configurations. Each row is further divided into two parts: firstly, the feature combinations including gradient components only (on the left), and secondly, all feature combinations based on the ELBCM descriptor (on the right). Within each cell, the different feature combinations are sorted by their number of features (n_F) in increasing order and by their F_1 scores median.

Figure 3 F_1-scores measuring the texture classification accuracy for the KTH-TIPS dataset using multiple feature combinations. Boxplots are used to highlight the concentration of F_1 scores and their median.

Three observations can be made from Figure 3: (1) that ELBCM-based combinations tend to have slightly higher scores than gradient-based ones (i.e., their F_1 scores are higher and more concentrated), (2) that color plays a crucial role to improve the discriminative power (purely gradient and ELBCM-based configurations both improve their scores when color information is included), and (3) the relevance of the spatial coordinates (x and y) seems to be small for the texture recognition problem.

According to Figure 3, the ideal number of features (among the set of feature combinations evaluated here) is between $n_F = 7$ and $n_F = 9$ given that most of the smaller configurations produce less accurate results. However, for such as size of covariance matrix, real-time execution (40 ms for 25 frames per second) as required for visual tracking is impossible without optimizations.

4 Covariance tracking algorithm analysis and optimizations

Three strategies are studied to optimize the CT on multi-core CPUs. The first one is based on the *structure of arrays* (SoA) towards *array of structures* (AoS) transformation: SoA→AoS. The second one consists in architectural optimizations: either multi-threading the SoA version with open multi-processing (OpenMP) middleware or using single instruction, multiple data (SIMD) instructions (SSE and AVX for Intel, Neon on ARM) for the AoS version. The third and final strategy consists of using loop-fusion transformations. In-depth information about the transformations employed in this article can be found in [23].

Let us introduce a set of notations for describing the algorithms and theirs optimizations (Table 2):

Table 2 Covariance descriptor algorithm notations

Notation	Meaning
I	Input image
h and w	Height and width of I
n_F	Number of features used to build the descriptor
n_P	Number of crossed-products of features $n_P = n_F(n_F+1)/2$
F	A data structure that contains all the features images
P	A data structure containing all the feature image products
I_F and I_P	The summed area tables (integral images) computed from F or P
$card$	the cardinal of SIMD register: 4 for SSE and Neon, 8 for AVX
vn_F	Number SIMD registers to hold n_F features: $vn_F = \lceil n_F/card \rceil$
vn_P	Number SIMD registers to hold n_P products: $vn_P = \lceil n_P/card \rceil$ (see Table 3)

In the baseline version of the algorithm, the complete set of feature images F is stored separately using a cube data structure (referred to as `fmat[k][i][j]`) which can be regarded as an instance of a SoA data structure. The index `k` is used here to select one of the n_F feature images while the pair (`i,j`) is used to select the spatial coordinates. Image cubes are straightforward to implement, the required arithmetic to compute the memory of an address using a table of 3D pointers only demands three integer additions; still, the latency time of a memory access is extremely dependent on the data access pattern.

4.1 SoA to AoS transform

The goal of SoA→AoS transform consists of transforming a set of independent arrays into one array, where each cell is a structure combining the elements of each independent array. The contribution of such a transform is to leverage the cache performance by enforcing spatial and temporal cache locality. Table 2 introduces the notations we will use from now on.

The first aspect we want to optimize is the locality of the features for a given point of coordinates (i, j). In the SoA version, we have two cubes: one that stores all the pixel features F_{SoA} (`fmat`) which size is $n_F \times h \times w$ and a different cube P_{SoA} (`prmat`) of size $n_P \times h \times w$ that stores the feature crossed-products. In the AoS data layout, these cubes are transformed into two 2D arrays F_{AoS} and P_{AoS} of size $h \times (w \cdot n_F)$ and $h \times (w \cdot n_P)$.

The SoA→AoS transform swaps the loop nests and changes the addressing computations from a 3D-form $cube[k][i][j]$ into a 2D-form like $matrix[i][j \times n + k]$, where n is the structure cardinal (here n_F or n_P). The lack of spatial locality within the features in the SoA representation is illustrated in Figure 4; here, the SoA layout (on the left) stores pixels features in discontiguous 2D arrays (distant in memory) while for the AoS representation (on the right) features belonging to the same pixel are gathered together in contiguous memory addresses.

Figure 4 Layout of SoA cube and the AoS feature matrix.

The covariance tracking algorithm is composed of three stages:

1. point-to-point product computation of all features,
2. the integral image computation of features,
3. the integral image computation of products.

The product of features and its transformation are described in Algorithms 1 and 2. Thanks to commutativity of the multiplication, only half of the products have to be computed (the loop on k_2 starts at k_1, line 3). As the two last stages are similar, we only present a generic version of integral image computation (Algorithm 3) and its transformation (Algorithm 4).

Algorithm 1: Product of features - SoA version

```
1 k ← 0
2 foreach k1 ∈ [0..nF − 1] do
3     foreach k2 ∈ [k1..nF − 1] do
4         foreach i ∈ [0..h − 1] do
5             foreach j ∈ [0..w − 1] do
6                 P[k][i][j] ← F[k1][i][j] × F[k2][i][j]
7                 k ← k + 1
```

Algorithm 2: Product of features - AoS version

```
1 foreach i ∈ [0..h − 1] do
2     foreach j ∈ [0..w − 1] do
3         k ← 0
4         foreach k1 ∈ [0..nF − 1] do
5             foreach k2 ∈ [k1..nF − 1] do
6                 P[i][j × nP + k] ← F[i][j × nP + k1] × F[i][j × nP + k2]
7                 k ← k + 1
```

Concerning the index k of Algorithms 1 and 2, the increment $k \leftarrow k + 1$ can be replaced by $k = k_1 n_F - k_1(k_1 + 1)/n + k_2$ for direct access to the product of feature k_1 by feature k_2.

Algorithm 3: Integral image - SoA version, $n \in \{n_F, n_P\}$

```
1 foreach k ∈ [0..n − 1] do
2     foreach i ∈ [0..h − 1] do
3         foreach j ∈ [0..w − 1] do
4             I[k][i][j] ← I[k][i][j] + I[k][i][j − 1] + I[k][i − 1][j] − I[k][i − 1][j − 1]
```

Algorithm 4: Integral image - AoS version, $n \in \{n_F, n_P\}$

```
1 foreach i ∈ [0..h − 1] do
2     foreach j ∈ [0..w − 1] do
3         foreach k ∈ [0..n − 1] do
4             I[i][j × n + k] ← I[i][j × n + k] + I[i][(j − 1) × n + k] + I[i − 1][j × n + k] − I[k][i − 1][(j − 1) × n + k]
```

4.2 SIMD or OpenMP parallelization?

Once this transform is done, one can also apply SIMD to the different parts of the algorithm. For the product part, the two internal loops on k_1 and k_2 are fully unrolled in order to show the list of all multiplications and the list of vectors to construct through permutation instructions (e.g., `_mm_shuffle_ps` in streaming SIMD extensions (SSE)). For example, for a typical value of $n_F = 7$, there are $n_P = 28$ products. The associated vectors are (the numbers are the feature indexes) as follows:

$$\begin{aligned}
[P_0, P_1, P_2, P_3] &= [F_0, F_0, F_0, F_0] \times [F_0, F_1, F_2, F_3] \\
[P_4, P_5, P_6, P_7] &= [F_0, F_0, F_0, F_1] \times [F_4, F_5, F_6, F_1] \\
[P_8, P_9, P_{10}, P_{11}] &= [F_1, F_1, F_1, F_1] \times [F_2, F_3, F_4, F_5] \\
[P_{12}, P_{13}, P_{14}, P_{15}] &= [F_1, F_2, F_2, F_2] \times [F_6, F_2, F_3, F_4] \\
[P_{16}, P_{17}, P_{18}, P_{19}] &= [F_2, F_2, F_3, F_3] \times [F_5, F_6, F_3, F_4] \\
[P_{20}, P_{21}, P_{22}, P_{23}] &= [F_3, F_3, F_4, F_4] \times [F_5, F_6, F_4, F_5] \\
[P_{24}, P_{25}, P_{26}, P_{27}] &= [F_4, F_5, F_5, F_6] \times [F_6, F_5, F_6, F_6]
\end{aligned}$$

In that case, the 7th vector is 100% filled, but it will become suboptimal if n_P is not divisible by the cardinal of the SIMD register (4 with SSE and Neon). In SSE, some permutations can be achieved using only one `_mm_shuffle_ps` instruction while others need a maximum of two. Because some permutations can be reused to perform other permutations, it is possible to achieve a factorization over all the required permutations. For example with $n_F = 7$, 15 shuffles are required.

In advanced vector extensions (AVX)2, there is a new instruction (compared to AVX) that greatly simplifies permutations : `_mm256_permutevar8x32_ps`. This instruction implements a full cross-bar, so we need exactly one AVX2 permutation per register that is a total 8 (for $n_F = 7$).

In Neon it is more complex. If some permutations can be done into 128-bits registers - that is with a parallelism of 4 - other permutations require instructions only available with 64-bit registers, like the *look-up table* instruction named `vtbl`. So in Neon, 128-bit float registers should be: 1) split into 64-bit registers with `vget_low_f32` and `vget_high_f32` instructions, 2) type-casted into 64-bit integer registers with `vreinterpret_u8_f32` – no latency, just for the compiler –, 3) permuted with `vtbl1_u8` and `vtbl2_u8` instructions, 4) type-casted into 64-bit float registers with `vreinterpret_f32_u8`, and 5) combined into 128-bit float registers with `vcombine_f32`. Finally it requires 31 SIMD Neon instructions to create the seven pairs of products (and 17 extra instructions for the castings). Table 3 gives the values of vn_F and vn_P depending on $n_F \in \{7, 8\}$ and $card$, the number of block within an SIMD register. For the same values of vn_F, Table 4 provides the number of permutations for SSE, AVX and Neon.

Table 3 Parameters values for scalar, 128-bit (SSE and Neon), and 256-bit (AVX) SIMD

Scalar		128-bit SIMD		256-bit SIMD	
n_F	n_P	vn_F	vn_P	vn_F	vn_P
7	28	2	7	1	4
8	36	2	9	1	5

Table 4 Permutation instructions for SSE, AVX, and Neon (permutation + cast) instruction set

n_F	SSE	AVX	Neon
7	15	8	31
8	17	10	35

The first part of Table 5 provides the algorithmic complexity and the amount of memory accesses for scalar version. Just replace n_F and n_P with vn_F and vn_P from Table 3 to get the SIMD value. This table also provides the arithmetic intensity (AI) - popularized by Nvidia - that is the ratio between the number of operations and the number of memory accesses. Table 6 provides numerical results from Table 5 for scalar, SSE, AVX, and Neon version; for 3-loop version; and for the 1-loop version with *loop-fusion* transform.

Table 5 Complexity, memory accesses, and arithmetic intensity of AoS versions with/without loop-fusion

Instructions	MUL	ADD	LOAD	STORE	AI
AoS version with 3 loops					
Product of features	n_P	0	$2n_P$	n_P	-
Integral of features	0	$3n_F$	$4n_F$	n_F	-
Integral of products	0	$3n_P$	$4n_P$	n_P	-
Total	n_P	$3(n_P + n_F)$	$6n_P + 4n_F$	$2n_P + n_F$	-
Total with $n_P = n_F(n_F+1)/2$		$2n_F^2 + 5n_F$		$4n_F^2 + 9n_F$	~ 0.5
AoS version + loop fusion with 1 loop					
Integral of features	0	$2n_F$	$2n_F$	n_F	-
Integral product of features	n_P	$2n_P$	n_P	n_P	-
Total	n_P	$2(n_P + n_F)$	$n_P + 2n_F$	$n_P + n_F$	-
Total with $n_P = n_F(n_F+1)/2$		$1.5n_F^2 + 3.5n_F$		$n_F^2 + 4n_F$	~ 1.5

Table 6 Complexity, memory accesses, and arithmetic intensity of scalar/SIMD versions with/without loop-fusion: numerical results for $n_F = 7$ and $n_F = 8$

Version		Without loop-fusion			With loop-fusion		
SIMD	n_F	Arith	Mem	AI	Arith	Mem	AI
Scalar	7	133	259	0.51	98	105	0.93
Scalar	8	168	328	0.51	124	132	0.94
SSE	7	49	66	0.74	40	27	1.48
SSE	8	59	82	0.72	48	33	1.45
Neon	7	65	66	0.98	56	27	2.07
Neon	8	77	82	0.94	66	33	2.00
AVX	7	27	37	0.73	22	15	1.47
AVX	8	33	45	0.73	27	18	1.50

For a given version, loop-fusion divides the complexity by 1.2 and memory accesses by 2.5.

Concerning OpenMP, the point is to evaluate *SOA + OpenMP* versus *AoS + SIMD*. Indeed, for a common 4-core General Purpose Processor (GPP), the degree of parallelism with a multi-threaded version and

with a SIMDized version is the same, i.e., four. Results are provided in cycles per point (*cpp*) *versus* the data amount (image size). The cpp is a normalized metric thats help to detect *cache overflow* (when data do not fit in the cache): the curve of cpp increases significantly.

The three versions (SoA + OpenMP, AoS, AoS + SIMD) have been benchmarked on three generations of Intel processors: Penryn, Nehalem, and SandyBridge for image sizes varying from 128×128 up to 1024×1024. It appears (Figure 5) that a 4-threaded version is always slower than a 1-threaded SIMD version. Eight threads are required on the Nehalem to be faster. The reason is the low AI inducing a high stress on the architecture's buses and also because manipulating SoA requires $n_P = 28$ active references in the cache; that is more than the usual L2 or L3 associativity (24 on the Intel processor). In the next steps of this article, SIMDization is the only architectural optimization being considered as realistic.

Figure 5 Performance in cpp of a 1×4-core Penryn (top), 2×4-core Nehalem (middle), 1×4-core SandyBridge (bottom) for image sizes $\in [128..1024]$.

4.3 Loop fusion

We have tested three versions with *loop-fusion* in order to increase the AI ratio by reducing the amount of memory accesses. But for that, we first have to rewrite the integral image computation. As integral image computation is known as being *memory bound*, but also a very simple algorithm (3 LOADs, 1 STORE, and 3 ADDs), it is quite impossible to reduce its complexity. Nevertheless, one can remove 2 LOADs by using a register that holds the accumulation along a line. Algorithm 5 implements this optimization for basic integral image computation.

Algorithm 5: Integral image - computation *in place* with an *accumulator*

```
1 x ← I[0][0], s ← x
2 foreach j ∈ [1..w − 1] do
3     x ← I[0][j], s ← s + x, I[0][j] ← s
4 foreach i ∈ [0..h − 1] do
5     x ← I[i][0], s ← x
6     foreach j ∈ [1..w − 1] do
7         x ← I[i][0], s ← s + x, I[i][j] ← s + I[i − 1][j]
```

The first one is a scalar parametric version (with n_F) that fuses the external `i`-loop and keeps the three `j`-loops unchanged. The second one is a specialized version with $n_F = 7$ where the three internal loops are fused together. The third one is the SIMDized version of the second one. The internal loop fusion allows to save the LOAD/STORE instructions in order to write a product of features into memory and to read it afterwards to compute the integral image of products. The loop-fusion has been done by hand, but some tools like PIPS [24] can do such a kind of transformation automatically [25]. The complexity of scalar and SIMD versions are provided in Table 5. The numerical value of these expressions is given in Table 6.

To be efficient loop-fusion is combined to full loop-unwinding (on k_1 and k_2) and scalarisation (to store temporary results within a register instead of a memory cell of an array). The behavior of the code is the following, for a given pixel (i, j):

- all the features associated to point (i, j) are loaded into n_F registers: $f_0, f_1, \cdots f_{n_F-1}$,
- the integral image computation of features is done on the fly and *in place* with Algorithm 5 with n_F accumulators $sf_0, sf_1 \cdots sf_{n_F-1}$. The point `fmat(i,j,k)` that previously holds n_F features is replaced by the sums stored in the n_F accumulator,

- the n_P products are then calculated in n_P registers: $p_{00}, p_{01}, ...p_{k_1k2}, \cdots p_{n_F-1nF-1}$
- the integral image computation of the product of features is done in the same way, with n_P accumulators. The point `prmat(i,j,k)` is filled with the n_P accumulators of products.

The code is quite big (as internal loops are unwound) but very efficient (see next section), but it can be easily generated automatically by a C program, as it is very systematic: load features do accumulation of features, store accumulations, and compute products and do accumulation of products and store accumulations. It is relevant for a bigger value of n_F to avoid bugs. The complexity of these new versions are given in the second part of Tables 5 and 6.

We can observe that without loop-fusion has the lowest AI of 0.5. We can notice that, for a given version, loop-fusion divides the complexity by a factor 1.2 (by rewriting image integral steps) and memory accesses by a factor 2.5 by avoiding LOADs and STOREs of temporary results.

4.4 Embedded systems

Let us now focus on more *embedded* processors like the Intel ULV (ultra low voltage) family and ARM processors. In order to observe the performance evolution for each family, two processors were benchmarked: Penryn-M U9300 (1.2GHz, 10 W, SSSE3), Haswell-M 4650U (1.7 GHz, 15 W, AVX2), ARM Cortex A9 (1.2 GHz, 1.2 W, Neon), and ARM Cortex A15 (1.7 GHz, 1.7 W, Neon). For Penryn-M and Haswell-M, the power consumption is the thermal dissipation power (TDP) provided by Intel; for ARM, these processors are part of SoC (Pandaboard OMAP4 from Texas Instruments and Samsung Chromebook with Exynos5 from Samsung) and it is very difficult to find out any figures from ARM or Samsung. So these figures were collected on the internet and cross-validated on *trustable* websites.

Figures 6 and 7 provide the cpp of the processor for image size varying from 64 to 1024 for Intel processors and from 64 to 512 for ARM processors.

Figure 6 Performance of Penryn-M and Haswell-M.

Figure 7 Performance of Cortex-A9 and Cortex-A15.

Firstly, for all processors, the SoA version is very inefficient compared to the best one (AoS+T+SIMD). The SIMDization alone is also inefficient: around $\times 1.5$ instead of $\times 4$ the ideal speedup for 128-bit SIMD and $\times 2.5$ instead of $\times 8$ for 256-bit SIMD. The reason is that SIMD is really efficient only (a speedup close to the SIMD register cardinal) when data fit in the cache [23]. Here the cache overflow appears for image size around 150×150 for ARM and 200×200 for Intel. As a matter of fact, a 512×512 image requires a cache of size of 36 MB, while a 640×480 needs 43 MB. If the biggest server processors just start to have such large cache (IBM Power7+, Intel Xeon Ivy bridge), such an amount of cache is far from the embedded ARM and Intel laptop processor (from 1 to 4 MB). The important fact, also common to the four processors, is that the cpp of AoS + T version remains constant, unlike SoA and AoS versions. So the execution time can be predicted.

Secondly, there is one big difference between them: the cpp values. The Intel cpp's are up to $\times 4.5$ smaller than ARM ones that comes from higher latency instructions.

Cortex-A15 is faster than A9 for two reasons: a faster cache and a faster external memory (same for Haswell-M versus Penryn-M) and because A15 can execute one Neon instruction every cycle instead of every two cycles: the SIMD throughput has been multiplied by two.

Regarding the impact of loop-fusion, Table 7 shows that the speedup with AoS version is from $\times 1.6$ up to $\times 1.7$ for ARM and Intel processor and for scalar and SIMD version, respectively. So the loop-fusion is as efficient as SIMDization. The total speedup is from $\times 3.8$ up to $\times 4.9$ for ARM and Penryn-M processor, respectively, but reaches $\times 7.9$ for Haswell-M (with SSE instructions).

Table 7 Impact of memory layout and loop-fusion transform

	Penryn-M	**Haswell-M**	**Cortex-A9**	**Cortex-A15**
cpp				
SoA	447	300	830	646
AoS	207	178	836	520
AoS + T	126	66	503	238
AoS + SIMD	165	69	476	325
AoS + T SIMD	92	38	201	169
speedups				
SoA/AoS	$\times 2.2$	$\times 1.7$	$\times 1.0$	$\times 1.2$
AoS/AoS + T	$\times 1.6$	$\times 2.7$	$\times 1.7$	$\times 2.2$
SIMD/SIMD + T	$\times 1.8$	$\times 1.8$	$\times 2.4$	$\times 1.9$
SoA/AoS+SIMD + T	$\times \mathbf{4.9}$	$\times \mathbf{7.9}$	$\times \mathbf{4.1}$	$\times \mathbf{3.8}$
Execution time (ms) for 512×512 images				
AoS+T SIMD	20.1	6.1	43.9	26.1
Estimated energy consumption (mJ) for 512×512 images				
AoS+T SIMD	201	91.4	52.7	44.3

cpp and speedups, execution time, and energy consumption for Penryn-M, Haswell-M, Cortex-A9 and Cortex-A15.

Concerning execution time, the Penryn-M and the Haswell-M are, respectively, $\times 4.3$ and $\times 2.2$ faster than the A9 and the A15. If we compare the estimation energy consumption (based on approximative TDP as previously said), the A9 and the A15 are, respectively, $\times 3.8$ and $\times 2.1$ more efficient than the Penryn-M and the Haswell-M. ARM embedded processors are still more efficient than Intel ones.

4.5 Impact of other parameters: SIMD width and n_F value

The impact of a twice wider SIMD - 256 bits for AVX2 instead of 128 for SSE - has been evaluated on a Haswell-M processor. It appears that there is quite no difference in performance between SSE and AVX2. First, AVX (and AVX2) processors can pair two SSE instructions within one AVX instruction, thanks to the *out-of-order* capabilities of these processors. Once the SIMD are fetched and decoded into the pipeline, they are put in the 'instructions ready' window before being dispatched to an execute unit (named *port* in Intel vocabulary). If the processor can find two SSE data-independent instructions that are ready to be executed, it pairs them together and sends the new instruction to an execute unit.

The impact of n_F has been also evaluated for the four processors. The two specialized scalar and SIMD versions AoS + T and AoS + T + SIMD have been instanced for $n_F = 8$ SSE, AVX, and Neon. It makes sense for AVX architecture as eight features 100% fills one AVX register (see Table 3). The cpp ratio (cpp($n_F = 8$)/cpp($n_F = 7$)) varies from 1.11 up to 1.35 for ARM processors and 1.21 up to 1.27 for Intel processors. These values are very close to the theoretical ratios (1.27 and 1.25) of the complexity and memory access amounts of Table 5. It means that the execution time of that part of the global algorithm is predictable until we run out of register and generate spill code.

5 Algorithm implementation

Two sequences have been used to evaluate the global performance on the four processors. Panda and Pedxing for which the robustness of the algorithm have been evaluated in [11] and [18]. For both of them, the execution times are given in cpp for each version of the algorithm: SoA is the basic version, and AoS++ stands for AoS transform + SIMDization + loop-fusion transform.

Two counter-intuitive results can be noticed. The first one is the features computation cpp: it is lower for SoA. The reason is obviously the memory layout of SoA (versus AoS) when computing the features and storing them into a cube or a matrix. The second counter-intuitive result is even more interesting: it concerns the tracking part of the algorithm which is based on the computation of a similarity criterion that requires the computation of the generalized eigenvalues, inversions, and matrix logarithms (9). In order to have the same behavior, we use GNU Scientific Library to perform these computations on both platforms, but we can also use Intel MKL or Eigen libraries. The future position is chosen by evaluating 40 (in our case, but it is parameterizable) random positions in a research window, so matrix operations represent a high percentage of the tracking part. It appears that the features used for tracking lead to a 'more' ill-conditioned matrix requiring more computations for Panda than for the Pedxing-3 sequence.

Concerning the acceleration, Tables 8 and 9 show that the optimization of the kernel provides a speedup of $\times 2.8$ to $\times 2.9$ for Intel processors and $\times 2.0$ to $\times 2.6$ for ARM ones that assets the need of all the optimizations.

Table 8 cpp and execution time for Intel Penryn-M and Haswell-M

Sequence	**Panda**		**Pedxing**	
Size	312×233		640×480	
Intel Penryn-M				
Algorithm version	SoA	AoS++	SoA	AoS++
Features computation (cpp)	128	150	128	150
Kernel computation (cpp)	599	87	618	91
Tracking (cpp)	23	23	11	11
Total (cpp)	738	248	769	264
Kernel/total	81%	35%	80%	34%
Total speedup	$\times 2.9$		$\times 2.8$	
1-C execution time (ms)	45	15	197	68
2-C execution time (ms)	36	9	158	38
Intel Haswell-M				
Algorithm version	SoA	AoS++	SoA	AoS++
Features computation (cpp)	78	79	88	72
Kernel computation (cpp)	190	36	207	40
Tracking (cpp)	13	23	2	3
Total (cpp)	281	138	297	115
Kernel/total	67%	26%	69%	34%
Total speedup	$\times 2.0$		$\times 2.6$	
1-C execution time (ms)	12	5	54	21
2-C execution time (ms)	8	3	37	13

Table 9 cpp and execution time for ARM Cortex-A9 and Cortex-A15

Sequence	**Panda**		**Pedxing**	
Size	312×233		640×480	
ARM Cortex-A9				
Algorithm version	SoA	AoS++	SoA	AoS++
Features computation (cpp)	461	461	486	486
Kernel computation (cpp)	1491	395	1600	415
Tracking (cpp)	96	96	19	19
Total (cpp)	2048	952	2106	921
Kernel/total	73%	42%	73%	45%
Total speedup	$\times 2.2$		$\times 2.3$	
1-C execution time (ms)	149	69	647	283
2-C execution time (ms)	108	36	492	149
ARM Cortex-A15				
Algorithm version	SoA	AoS++	SoA	AoS++
Features computation (cpp)	207	207	205	205
Kernel computation (cpp)	562	170	582	177
Tracking (cpp)	28	52	4	7
Total (cpp)	797	429	791	389
Kernel/total	70%	39%	73%	45%
Total speedup	$\times 1.9$		$\times 2.0$	
1-C execution time (ms)	38	20	161	79
2-C execution time (ms)	27	10	119	42

As both processors have two cores, all the processing parts can be done either on one core (the execution time is the sum of all parts) or on two cores (the biggest part is on one core and the two other parts are on the second core). With such a coarse grain thread distribution, the Penryn-M and the Haswell-M can track targets in real time for 640×480 images. The Haswell-M is even real time with only one core. The Cortex-A9 can do it for image sizes up to 320×240 and the Cortex-A15 is close to real time for 640×480 images. Once the kernel computation has been optimized, the biggest processing part becomes the features computation. With the optimization of this part, the Cortex-A15 should be able to reach real-time execution.

The performance ratio of the whole algorithm is close to the performance ratio of the kernel: the Penryn-M and the Haswell-M are, respectively, $\times 4.0$ and $\times 3.3$ faster than the Cortex-A9 and the Cortex-A15. We can also observe that the image size has quite no impact on the performance ratio. From an energy point of view, the Cortex-A9 and the Cortex-A15 are, respectively, $\times 2.1$ and $\times 2.7$ more energy efficient than the Penryn-M and the Haswell-M.

6 Conclusions

We have presented the implementation of a robust covariance tracking algorithm, with a parameterizable complexity that can be adapted to trade-off between robustness and execution time. A study has been made to qualitatively compare different covariance matrices in terms of number and nature of visual features. Classical software and hardware optimizations have been applied: SIMDization and loop-fusion transform combined with AoS-SoA transform to accelerate the kernel of the algorithm. These optimizations allow a real-time execution (25 fps or about 40 ms per image) for 320×240 image size on ARM Cortex-A9 and for 640×480 on Intel Penryn-M and Haswell. ARM Cortex-A15 should also reach real-time execution for such image size, once the other parts of the algorithm will be optimized.

Our future work will focus on (1) the optimization of the features computation and (2) the multi-threading of the tracking in order to perform multi-target tracking with load balancing on the available core. A more thorough study should also be made concerning the impact of the ill-conditioning of the matrix on the execution time.

To the best of our knowledge, our implementation of the covariance tracking algorithm is the first real-time implementation for embedded systems, while perfectly maintaining the quality of the tracking.

Endnotes

[a]K-Nearest Neighbours

Competing interests

The authors declare that they have no competing interests.

References

1. Z Kalal, K Mikolajczyk, J Matas, Tracking-learning-detection. Pattern Anal. Mach. Intell. IEEE Trans. **34**(7), 1409–1422 (2012)
2. BD Lucas, T Kanade, An iterative image registration technique with an application to stereo vision. in *Proceedings of the 7th International Joint Conference on Artificial Intelligence* (Morgan Kaufmann, San Francisco, 1981), pp. 674–679
3. D Comaniciu, V Ramesh, P Meer, Kernel-based object tracking. Pattern Anal. Mach. Intell. IEEE Trans. **25**(5), 564–577 (2003)
4. M Gouiffès, F Laguzet, L Lacassagne, Color connectedness degree for mean-shift tracking. in *Pattern Recognition (ICPR), 2010 20th International Conference On* (IEEE, 2010), pp. 4561–4564
5. F Porikli, O Tuzel, P Meer, Covariance tracking using model update based on lie algebra. in *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference On*, vol. 1 (IEEE, 2006), pp. 728–735
6. P Li, Q Wang, *Local Log-Euclidean Covariance Matrix (L2ECM) for Image Representation and Its Applications* (Springer, 2012)
7. Y Zhang, S Li, Gabor-LBP based region covariance descriptor for person re-identification. in *Image and Graphics (ICIG), 2011 Sixth International Conference on* (2011), pp. 368–371
8. S Guo, Q Ruan, Facial expression recognition using local binary covariance matrices. in *Wireless, Mobile & Multimedia Networks (ICWMMN 2011), 4th IET International Conference on* (2011), pp. 237–242
9. Y Pang, Y Yuan, X Li, Gabor-based region covariance matrices for face recognition. Circuits Syst. Video Technol. IEEE Trans. **18**(7), 989–993 (2008)
10. S Bak, E Corvee, F Bremond, M Thonnat, Multiple-shot human re-identification by mean Riemannian covariance grid. in *Advanced Video and Signal-Based Surveillance (AVSS), 2011 8th IEEE International Conference On* (IEEE, 2011), pp. 179–184
11. A Romero, M Gouiffès, L Lacassagne, Covariance descriptor multiple object tracking and re-identication with colorspace evaluation. in *Asian Conference on Computer Vision, 2012. ACCV 2012* (2012)

12. Y Wu, B Wu, J Liu, H Lu, Probabilistic tracking on Riemannian manifolds. in *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on* (2008), pp. 1–4

13. A Tyagi, JW Davis, G Potamianos, Steepest descent for efficient covariance tracking. in *Motion and Video Computing, 2008. WMVC 2008. IEEE Workshop on* (2008), pp. 1–6

14. X Zhang, G Dai, N Xu, Genetic algorithms. A new optimization and search algorithms. Control Theory Appl. **3**, (1995)

15. A Romero, L Lacassagne, M Gouiffes, Real-time covariance tracking algorithm for embedded systems. in *IEEE International Conference on Design and Architectures for Signal and Image Processing (DASIP)* (2013)

16. O Tuzel, F Porikli, P Meer, Pedestrian detection via classification on Riemannian manifolds. Pattern Anal. Mach. Intell. IEEE Trans. **30**(10), 1713–1727 (2008)

17. J Yao, JM Odobez, Fast human detection from videos using covariance features. in *The Eighth International Workshop on Visual Surveillance-VS2008* (2008)

18. A Romero, M Gouiffès, L Lacassagne, Enhanced local binary covariance matrices ELBCM for texture analysis and object tracking. in *ACM International Conference Proceedings Series* (Association for Computing Machinery, 2013)

19. M Pietikäinen, A Hadid, G Zhao, T Ahonen, *Computer Vision Using Local Binary Patterns* (Springer, 2011). http://books.google.fr/books?id=wBrZz9FiERsC

20. O Tuzel, F Porikli, P Meer, Region covariance: a fast descriptor for detection and classification. Computer Vision–ECCV 2006, **3952**, 589–600 (2006)

21. W Förstner, B Moonen, A metric for covariance matrices. Quo Vadis Geodesia, 113–128 (1999)

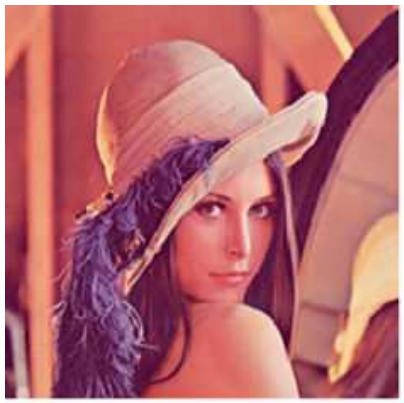
22. E Hayman, B Caputo, M Fritz, J-O Eklundh, On the significance of real-world conditions for material classification. Computer Vision-ECCV 2004, **3024**, 253–266 (2004)

23. L Lacassagne, D Etiemble, A Hassan-Zahraee, A Dominguez, P Vezolle, High level transforms for SIMD and low-level computer vision algorithms. in *ACM Workshop on Programming Models for SIMD/Vector Processing (PPoPP)* (2014), pp. 49–56

24. MINES-ParisTech, PIPS. http://pips4u.org. Open source, under GPLv3 (1989–2009)

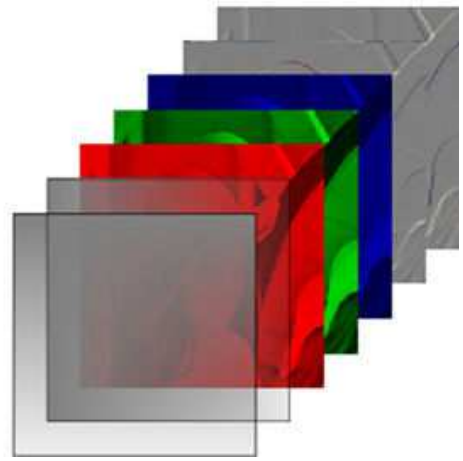
25. F Irigoin, P Jouvelot, R Triolet, Semantical interprocedural parallelization: an overview of the PIPS project. in *ICS '91 Proceedings of the 5th international conference on Supercomputing* (ACM, New York, 1991), pp. 244–251

Input Image $I(x, y)$

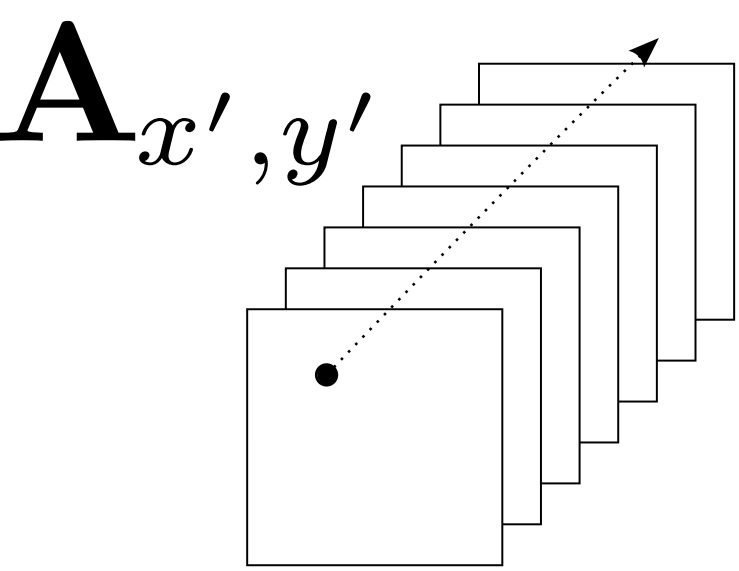


Feature Image Tensor

$F(x, y) = \phi(I, x, y)$



Tensor of Integral Images



Second order Integral Images Tensor

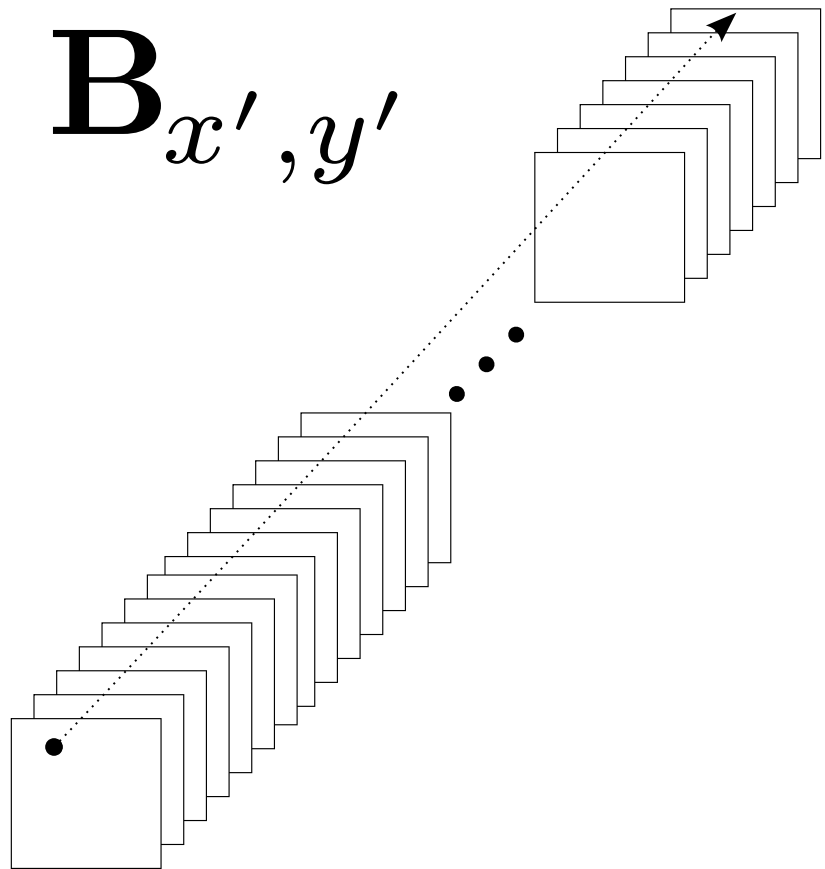


Figure 1

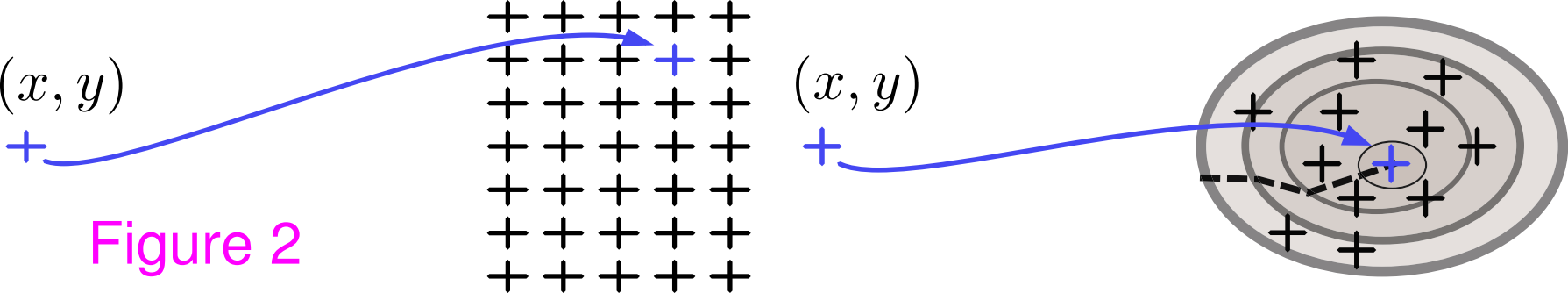
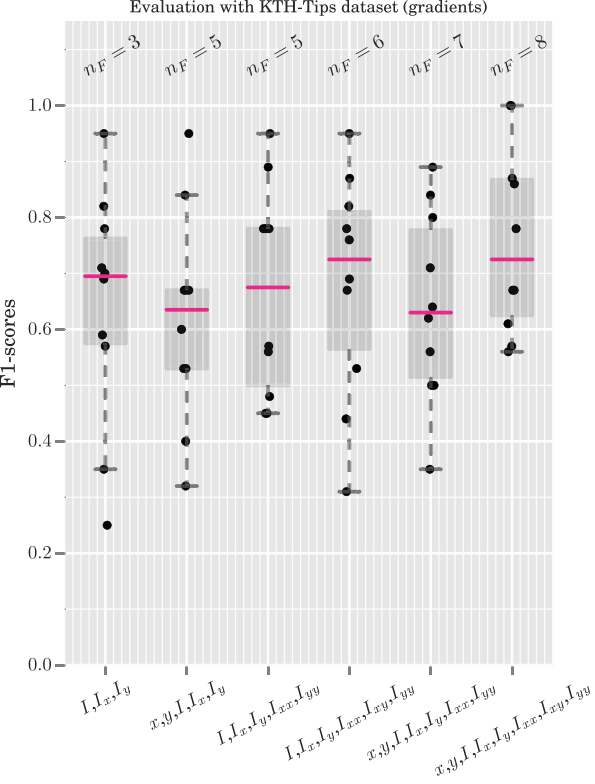
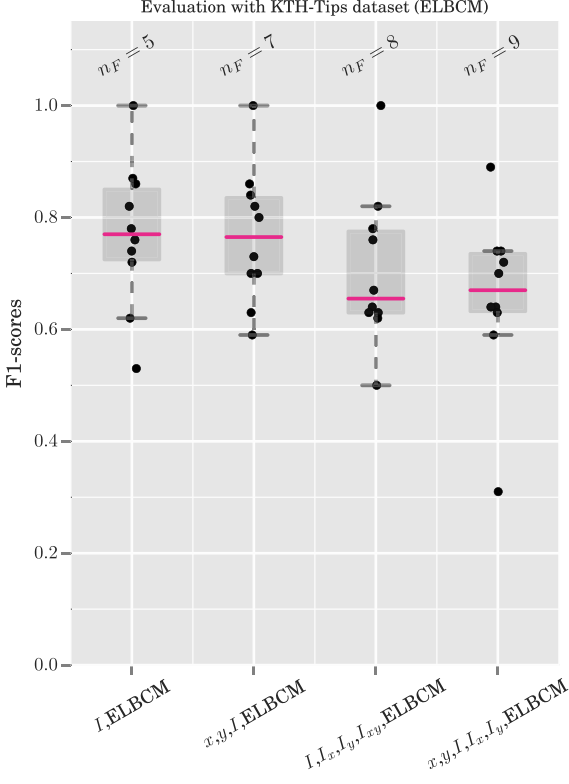
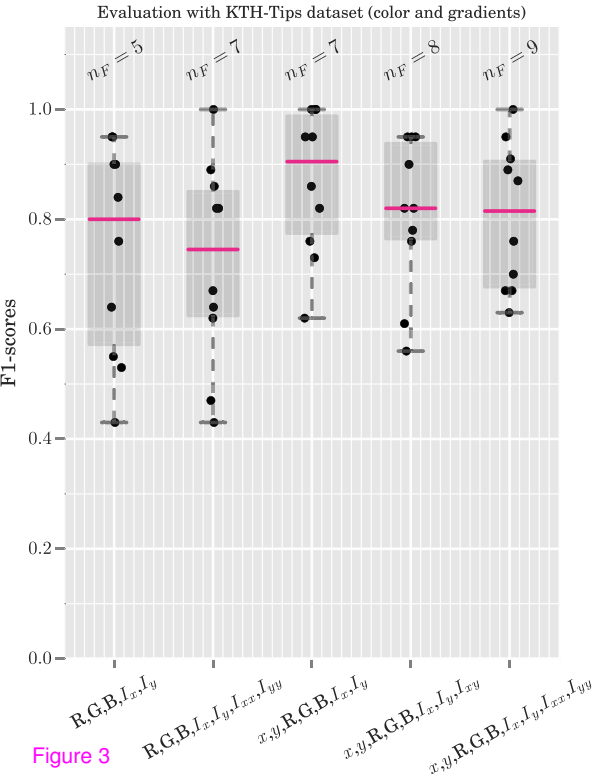


Figure 2


Evaluation with KTH-Tips dataset (gradients)
$n_F = 3$
$n_F = 5$
$n_F = 5$
$n_F = 6$
$n_F = 7$
$n_F = 8$
F1-scores
1.0
0.8
0.6
0.4
0.2
0.0
I, I_x, I_y
x, y, I, I_x, I_y
$I, I_x, I_y, I_{xx}, I_{yy}$
$I, I_x, I_y, I_{xx}, I_{xy}, I_{yy}$
$x, y, I, I_x, I_y, I_{xx}, I_{yy}$
$x, y, I, I_x, I_y, I_{xx}, I_{xy}, I_{yy}$


Evaluation with KTH-Tips dataset (ELBCM)
$n_F = 5$
$n_F = 7$
$n_F = 8$
$n_F = 9$
F1-scores
1.0
0.8
0.6
0.4
0.2
0.0
I,ELBCM
x, y, I,ELBCM
I, I_x, I_y, I_{xy},ELBCM
x, y, I, I_x, I_y,ELBCM


Evaluation with KTH-Tips dataset (color and gradients)
$n_F = 5$
$n_F = 7$
$n_F = 7$
$n_F = 8$
$n_F = 9$
F1-scores
1.0
0.8
0.6
0.4
0.2
0.0
R,G,B,I_x, I_y
R,G,B,I_x, I_y, I_{xx}, I_{yy}
x, y,R,G,B,I_x, I_y
x, y,R,G,B,I_x, I_y, I_{xy}
x, y,R,G,B,I_x, I_y, I_{xx}, I_{yy}

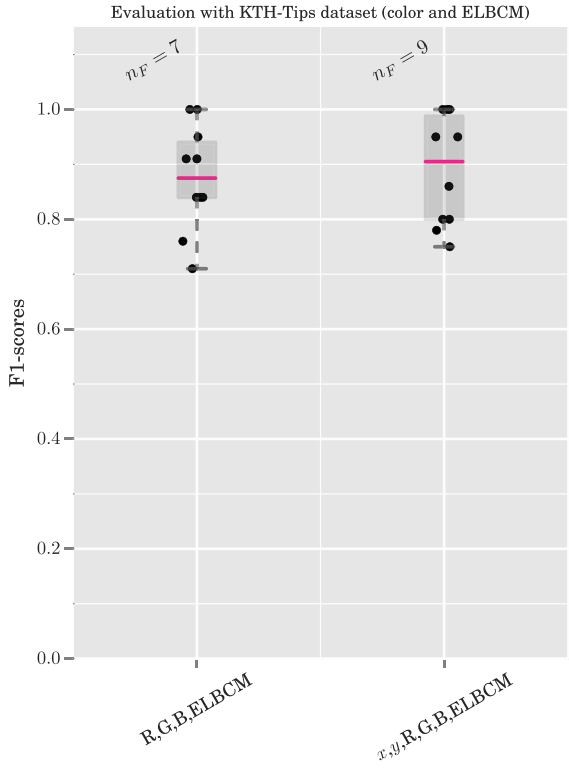

Evaluation with KTH-Tips dataset (color and ELBCM)
$n_F = 7$
$n_F = 9$
F1-scores
1.0
0.8
0.6
0.4
0.2
0.0
R,G,B,ELBCM
x, y,R,G,B,ELBCM

Figure 3

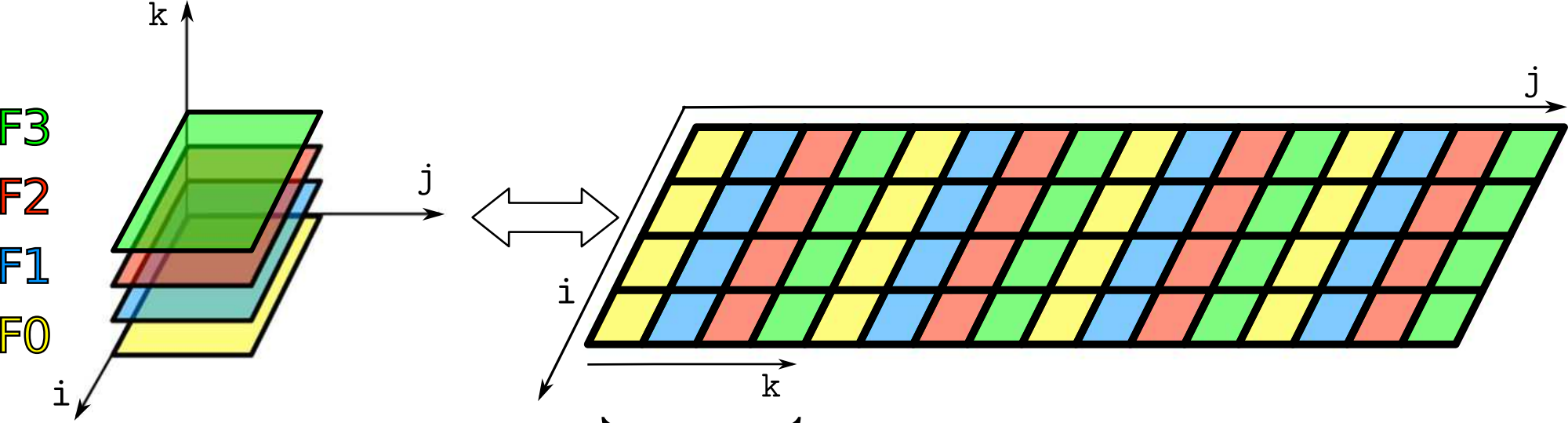


Figure 4

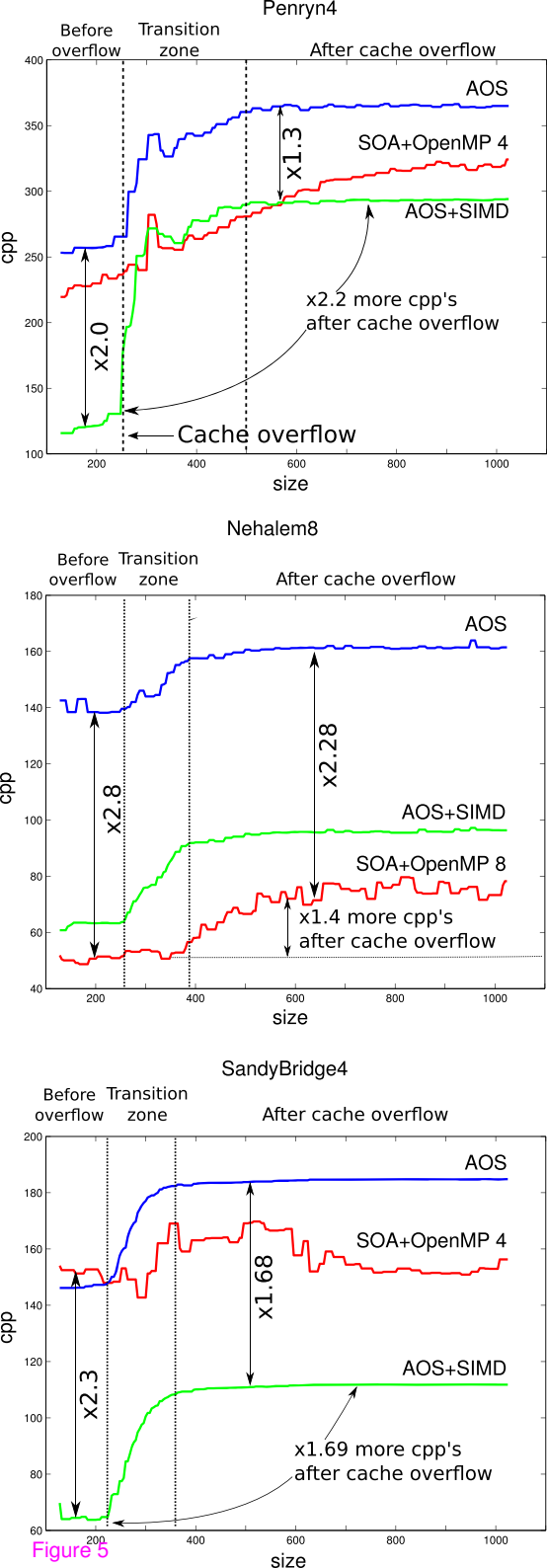
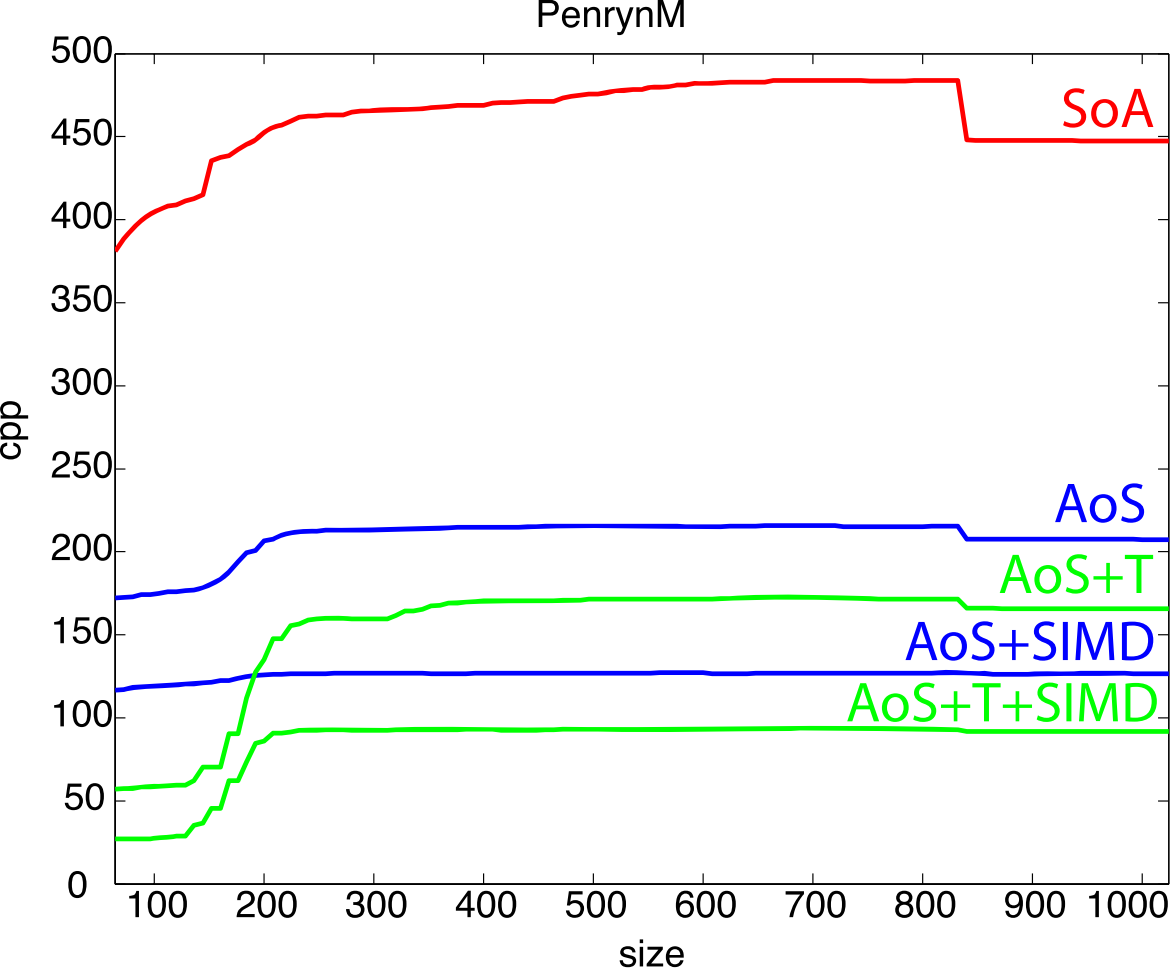

Penryn4
Before overflow
Transition zone
After cache overflow
AOS
SOA+OpenMP 4
AOS+SIMD
x1.3
x2.0
x2.2 more cpp's after cache overflow
Cache overflow
cpp
size
Nehalem8
Before overflow
Transition zone
After cache overflow
AOS
AOS+SIMD
SOA+OpenMP 8
x2.8
x2.28
x1.4 more cpp's after cache overflow
cpp
size
SandyBridge4
Before overflow
Transition zone
After cache overflow
AOS
SOA+OpenMP 4
AOS+SIMD
x2.3
x1.68
x1.69 more cpp's after cache overflow
cpp
size

Figure 5


PenrynM
500
450
400
350
300
250
200
150
100
50
0
cpp
SoA
AoS
AoS+T
AoS+SIMD
AoS+T+SIMD
100
200
300
400
500
600
700
800
900
1000
size

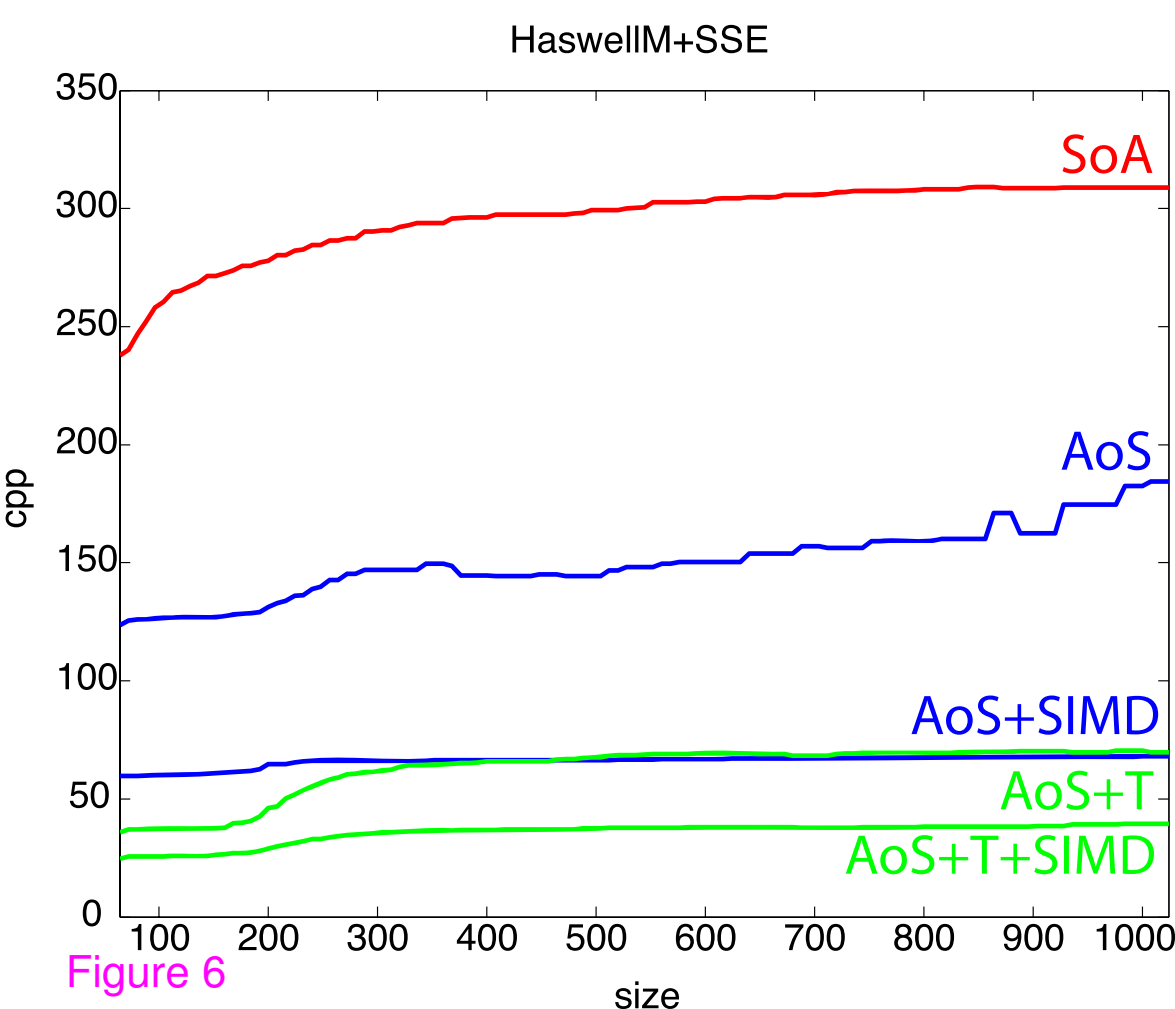
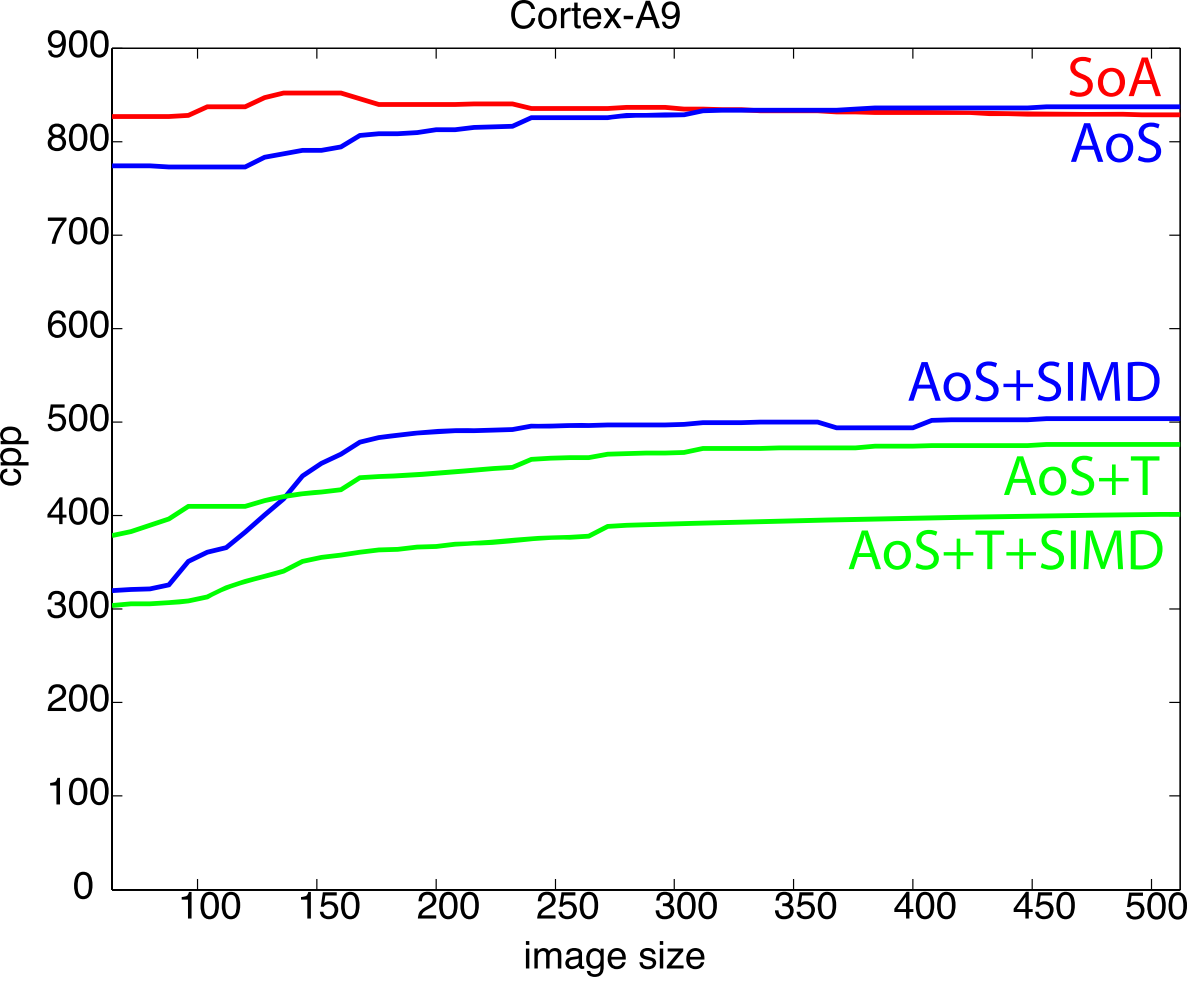

HaswellM+SSE
350
300
250
200
150
100
50
0
cpp
SoA
AoS
AoS+SIMD
AoS+T
AoS+T+SIMD
100
200
300
400
500
600
700
800
900
1000
size

Figure 6


Cortex-A9
900
800
700
600
500
400
300
200
100
0
cpp
SoA
AoS
AoS+SIMD
AoS+T
AoS+T+SIMD
100
150
200
250
300
350
400
450
500
image size

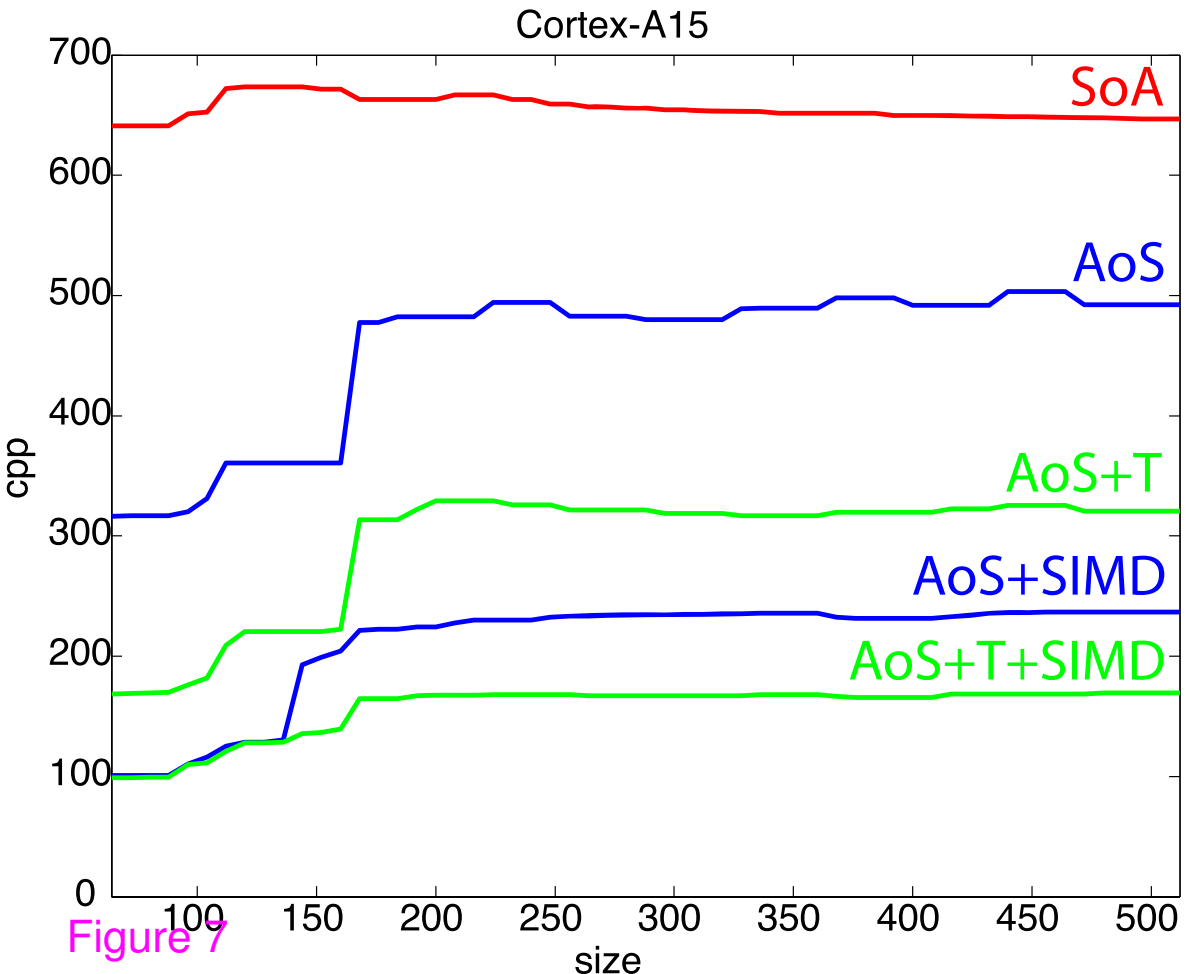

Cortex-A15
700
600
500
400
300
200
100
0
cpp
SoA
AoS
AoS+T
AoS+SIMD
AoS+T+SIMD
100
150
200
250
300
350
400
450
500
size

Figure 7